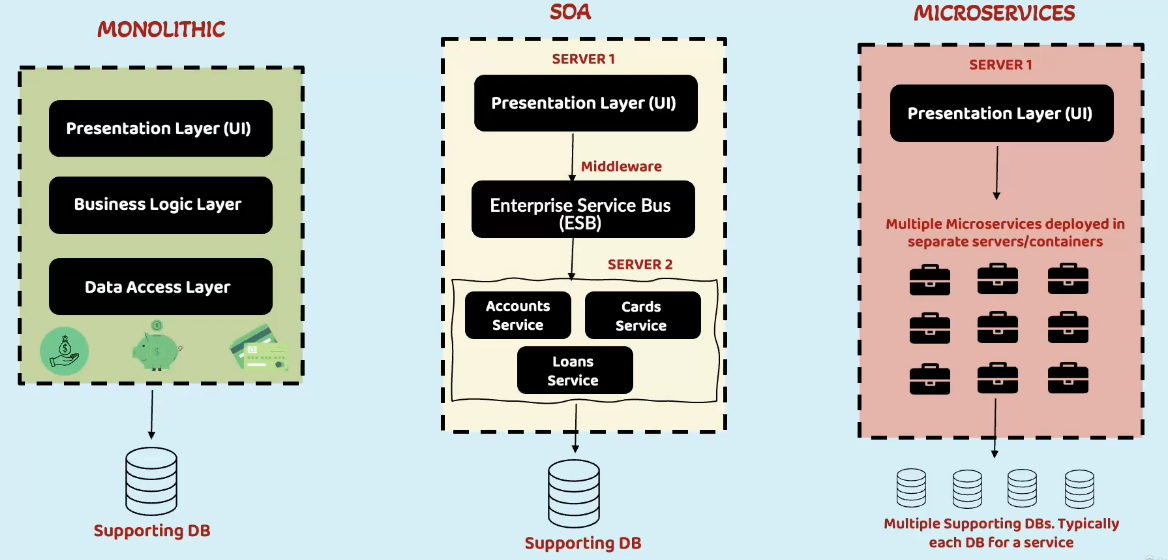
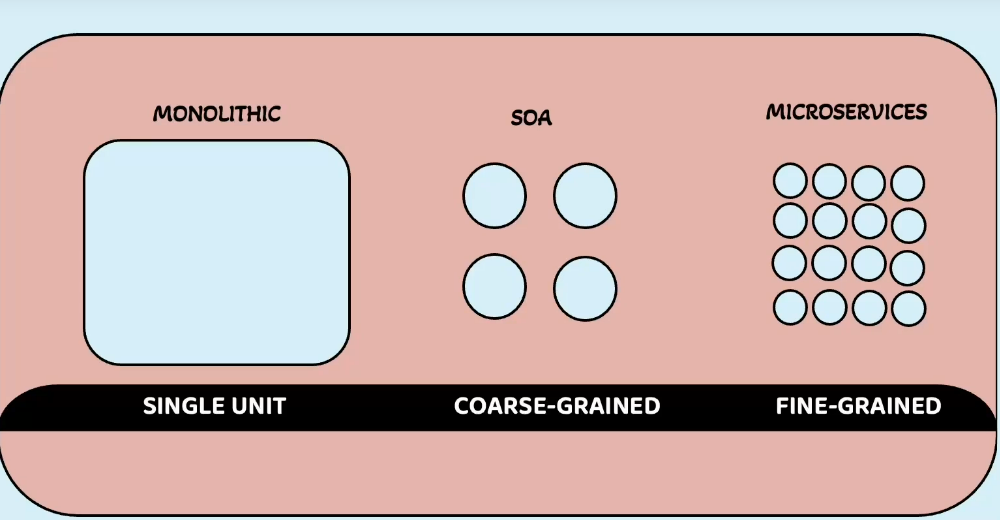
**Section 1  
Difference between Monolithic, SOA and Microservices - INTERVIEW**

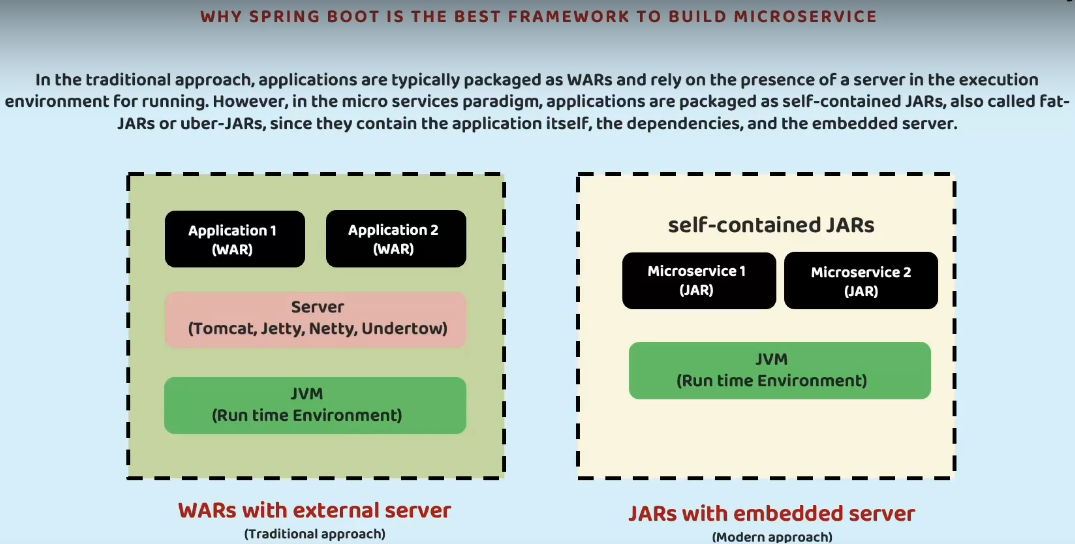
Monolithic:   
All applications will be deployed in a single server, supported by a single database.  
The web application and the business logic is tightly coupled.  
  
SOA:  
We separate the UI and the backend logic in different servers, but again supported by a single database.   
We add a middle ware extra component (Enterprise Service Bus - ESB) which is difficult to maintain.  
There is some level of separation but not based on business domain.  
  
Microservice:  
Here we create separate deployable services based on business domain and each has its own database. All these microservices and databases can be a mix of technology.   
So, each service has its own separate development, code base and deployment lifecycle.  
  


|  |  |  |  |
| --- | --- | --- | --- |
| **Features** | **Monolithic** | **SOA** | **Microservices** |
| Parallel development by teams on features which are component specific | NO | MIDDLE (separation between UI and backend teams) | YES |
| Agility (Enhancing the application or component with new language / framework) | NO | MIDDLE (separation between UI and backend teams) | YES |
| Scalability | NO - Not possible because you are going to deploy all your application in one single jumbo server, so to scale you bring in one more jumbo server and need to take care of load balancing – manual setup | MIDDLE – Difficult to scale you, as are going to deploy all your application in one single jumbo server. | YES – Scalability is easy, because of products like docker and Kubernetes. |
| Usability | NO | MIDDLE | YES – Any new feature specific for domain component can be deployed without any interference to any other component. |
| Complexity and Operational Overhead | YES – Here there is only one server, you have to make sure just that one server is running without any issues | MIDDLE – Here you need to take care of UI server, middle ware server and backend server. | NO |
| Security and Performance | YES – Here there use to be method calls | MIDDLE | NO – Here the communication between components/service are REST services, so request goes over the network. So there may be some network latency. |

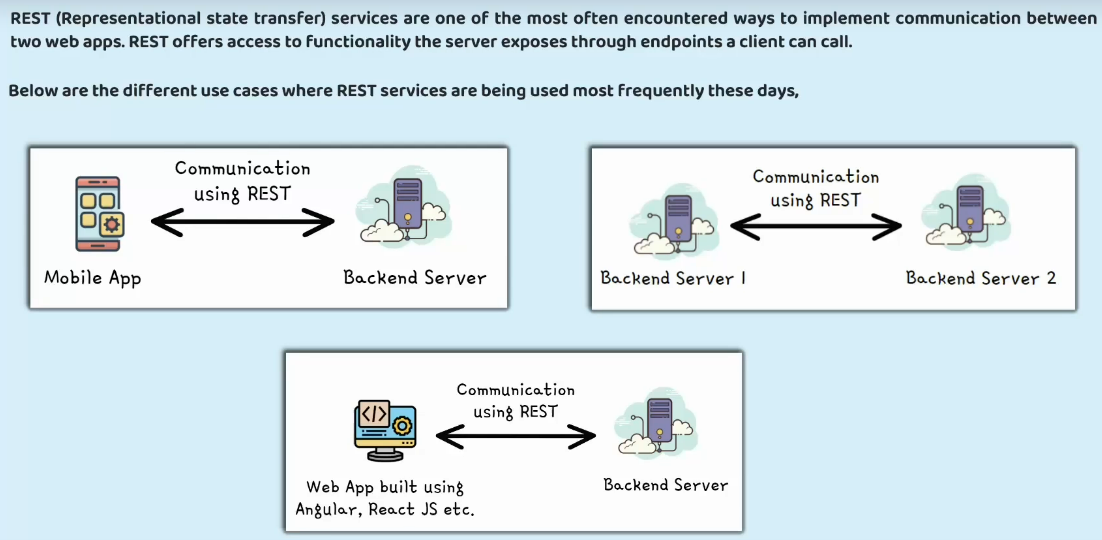
**Definition of Microservices**

**Section 2**  

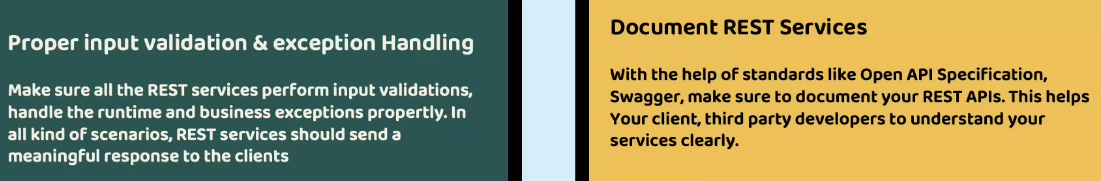

****

****

**Introduction to REST API and best practices**

****

****

****

**Creating Spring Boot project**<parent>

        <groupId>org.springframework.boot</groupId>

        <artifactId>spring-boot-starter-parent</artifactId>

        <version>3.1.5</version>

</parent>

<groupId>com.eazybytes</groupId>

<artifactId>accounts</artifactId>

<version>0.0.1-SNAPSHOT</version>

<dependencies>

        <dependency>

            <groupId>org.springframework.boot</groupId>

            <artifactId>spring-boot-starter-actuator</artifactId>

        </dependency>

        <dependency>

            <groupId>org.springframework.boot</groupId>

            <artifactId>spring-boot-starter-data-jpa</artifactId>

        </dependency>

        <dependency>

            <groupId>org.springframework.boot</groupId>

            <artifactId>spring-boot-starter-validation</artifactId>

        </dependency>

        <dependency>

            <groupId>org.springframework.boot</groupId>

            <artifactId>spring-boot-starter-web</artifactId>

        </dependency>

        <dependency>

            <groupId>org.springframework.boot</groupId>

            <artifactId>spring-boot-devtools</artifactId>

            <scope>runtime</scope>

            <optional>true</optional>

        </dependency>

        <dependency>

            <groupId>com.h2database</groupId>

            <artifactId>h2</artifactId>

            <scope>runtime</scope>

        </dependency>

        <dependency>

            <groupId>org.projectlombok</groupId>

            <artifactId>lombok</artifactId>

            <optional>true</optional>

        </dependency>

    </dependencies>

import org.springframework.boot.autoconfigure.SpringBootApplication;  
With this annotation we are spring boot framework to perform auto configurations and to scan all the beans inside the spring boot application.  
So SpringBootApplication is a combination of all these annotations   
@Inherited  
@SpringBootConfiguration  
@EnableAutoConfiguration  
@ComponentScan  
  
**Creating Hello World REST API using @RestController**   
Whenever we are putting an annotation on top of a class we are instructing the framework, that I am going to write method in this class related to http methods, so expose all these methods as a REST API to the outside world.  
  
package com.eazybytes.accounts.controller;

@RestController

public class AccountsController {

@GetMapping("sayHello")

public String sayHello(){

return "Say Hello";

}

}

**Configuring H2 DB and yaml properties**accounts\src\main\resources\application.properties rename to application.yml  
  
spring.datasource.url.=jdbc:h2:mem:testdb

spring.datasource.driverClassName=org.h2.Driver

spring.datasource.username=sa

spring.datasource.password=

spring.jpa.database-platform=org.hibernate.dialect.H2Dialect

spring.h2.console.enabled=true

spring.jpa.hibernate.ddl-auto=update

spring.jpa.show-sql=true

server:

port: 8080

spring:

datasource:

url: jdbc:h2:mem:testdb

driver-class-name: org.h2.Driver

username: sa

password:

h2:

console:

enabled: true

jpa:

database-platform: org.hibernate.dialect.H2Dialect

hibernate:

ddl-auto: update

show-sql: true

accounts\src\main\resources\schema.sql  
CREATE TABLE IF NOT EXISTS `customer` (

`customer\_id` int AUTO\_INCREMENT PRIMARY KEY,

`name` varchar(100) NOT NULL,

`email` varchar(100) NOT NULL,

`mobile\_number` varchar(20) NOT NULL,

`created\_at` date NOT NULL,

`created\_by` varchar(20) NOT NULL,

`updated\_at` date DEFAULT NULL,

`updated\_by` varchar(20) DEFAULT NULL

);

CREATE TABLE IF NOT EXISTS `accounts` (

`customer\_id` int NOT NULL,

`account\_number` int AUTO\_INCREMENT PRIMARY KEY,

`account\_type` varchar(100) NOT NULL,

`branch\_address` varchar(200) NOT NULL,

`created\_at` date NOT NULL,

`created\_by` varchar(20) NOT NULL,

`updated\_at` date DEFAULT NULL,

`updated\_by` varchar(20) DEFAULT NULL

);

Start the spring application  
<http://localhost:8080/h2-console/>

You will see that the 2 tables are created ‘customer’ and ‘accounts’.  
  
**Writing Spring Data JPA Entities and repositories to interact with DB tables**

For this we have already added the library spring-boot-starter-data-jpa in which we have many interfaces and classes which we can use to interact with the database.  
  
Before using this framework, we need to create a POJO or entity classes which represent the tables in our database.  
  
CREATE TABLE IF NOT EXISTS `customer` (

`customer\_id` int AUTO\_INCREMENT PRIMARY KEY,

`name` varchar(100) NOT NULL,

`email` varchar(100) NOT NULL,

`mobile\_number` varchar(20) NOT NULL,

**`created\_at` date NOT NULL,**

**`created\_by` varchar(20) NOT NULL,**

**`updated\_at` date DEFAULT NULL,**

**`updated\_by` varchar(20) DEFAULT NULL**

);

CREATE TABLE IF NOT EXISTS `accounts` (

`customer\_id` int NOT NULL,

`account\_number` int AUTO\_INCREMENT PRIMARY KEY,

`account\_type` varchar(100) NOT NULL,

`branch\_address` varchar(200) NOT NULL,

**`created\_at` date NOT NULL,**

**`created\_by` varchar(20) NOT NULL,**

**`updated\_at` date DEFAULT NULL,**

**`updated\_by` varchar(20) DEFAULT NULL**

);

There are 4 columns which are common to each of these tables. These are called metadata columns. So, these should have a separate super class BaseEntity.

`created\_at` date NOT NULL, so the field name should be createdAt  
`created\_by` varchar(20) NOT NULL, so the field name should be createdBy  
So no need to mention @Column(name=”created\_at”) for fields

By doing this we do not have the need to tell the framework which column this field maps to.  
  
@MappedSuperclass this signify that this entity class will act the super class for all you entity classes.  
  
@Column(updatable = false) when the row is being updated we don’t want this column to be considered by spring data JPA for updating, this fields will not be updated when ever we update the record. It will be populated only once when the record is inserted or created.  
  
@Column(insertable = false) this tells do not populate this column when ever a new record is inserted or created.  
  
package com.eazybytes.accounts.entity;

import java.time.LocalDateTime;

import jakarta.persistence.Column;

import jakarta.persistence.MappedSuperclass;

import lombok.Getter;

import lombok.Setter;

import lombok.ToString;

@MappedSuperclass

@Getter @Setter @ToString

public class BaseEntity {

@Column(updatable = false)

private LocalDateTime createdAt;

@Column(updatable = false)

private String createdBy;

@Column(insertable = false)

private LocalDateTime updatedAt;

@Column(insertable = false)

private String updatedBy;

}

@Entity this is telling the JPA framework please treat this class as entity or POJO representation of the table ‘customer’. Table name should match with the class name Customer.

If it does not match you can use annotation @Table(name=”customer”) on top of the Entity class.  
  
@AllArgsConstructor @NoArgsConstructor these are required when we try to create the object of this class.

@Id is used to signify the primary key of the table, but   
are we going to provide the value to the primary key manually   
or  
we want the spring data framework to generate the value by itself when we try to insert a new record inside the table.  
  
Here in this case its good to give the responsibility to the spring data framework. For this   
@GeneratedValue(strategy = GenerationType.AUTO,generator = "native")

@GenericGenerator(name = "native", strategy = "native")  
  
this tells the spring data framework generate the value automatically, With “native” strategy we are telling spring data framework, whatever database that I am using please try to generate the sequence number or the primary key value based upon the native style of my database.

package com.eazybytes.accounts.entity;

import org.hibernate.annotations.GenericGenerator;

import jakarta.persistence.Column;

import jakarta.persistence.Entity;

import jakarta.persistence.GeneratedValue;

import jakarta.persistence.GenerationType;

import jakarta.persistence.Id;

import lombok.AllArgsConstructor;

import lombok.Getter;

import lombok.NoArgsConstructor;

import lombok.Setter;

import lombok.ToString;

@Entity

@Getter @Setter @ToString @AllArgsConstructor @NoArgsConstructor

public class Customer extends BaseEntity{

@Id

@GeneratedValue(strategy = GenerationType.AUTO,generator = "native")

@GenericGenerator(name = "native", strategy = "native")

@Column(name = "customer\_id")

private Long customerId;

private String name;

private String email;

@Column(name = "mobile\_number")

private String mobileNumber;

}

For accountNumber which is the primary key we will have our own logic to generate the value. Because we don’t account number to start with 1, 2, etc.. account numbers are usually 10 digit numbers.  
  
package com.eazybytes.accounts.entity;

import jakarta.persistence.Column;

import jakarta.persistence.Entity;

import jakarta.persistence.Id;

import lombok.AllArgsConstructor;

import lombok.Getter;

import lombok.NoArgsConstructor;

import lombok.Setter;

@Entity

@Getter @Setter @AllArgsConstructor @NoArgsConstructor

public class Accounts extends BaseEntity{

@Id

@Column(name = "account\_number")

private Long accountNumber;

@Column(name = "account\_type")

private String accountType;

@Column(name = "branch\_address")

private String branchAddress;

}

Now we should also create the repository classes.  
Create package com.eazybytes.accounts.repository  
@Repository when we mention this annotation, spring data JPA framework will create the bean implementation (runtime code having lots of methods for various sorts of CRUD operations) of this interface in the background based on the configurations that we provided in application.yml and this interface.  
  
extends JpaRepository<Customer, Long> so this represents which is the entity class which is going to be handled by this repository class and what is the datatype of your primary key.  
JpaRepository internally extends various interfaces ListCrudRepository, ListPagingAndSortingRepository, QueryByExampleExecutor   
  
package com.eazybytes.accounts.repository;

import org.springframework.data.jpa.repository.JpaRepository;

import org.springframework.stereotype.Repository;

import com.eazybytes.accounts.entity.Customer;

@Repository

public interface CustomerRepository extends JpaRepository<Customer, Long>{

}

package com.eazybytes.accounts.repository;

import org.springframework.data.jpa.repository.JpaRepository;

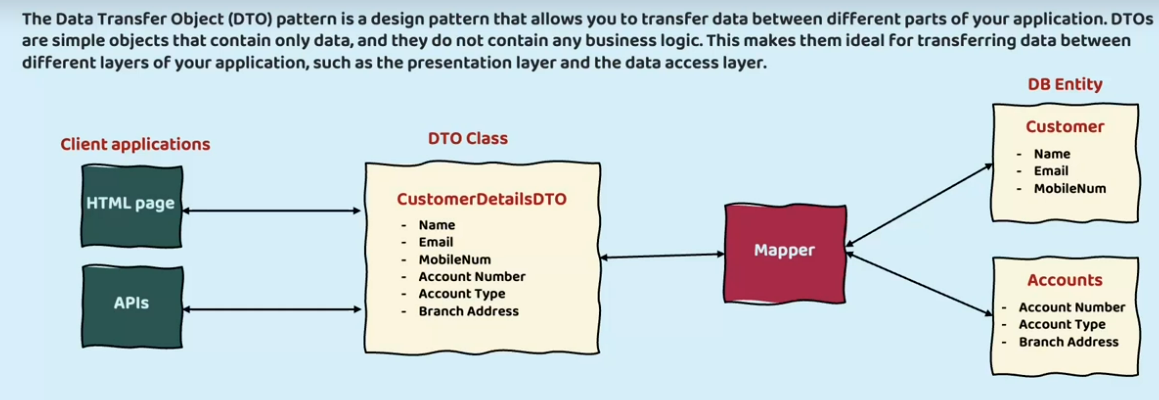
import org.springframework.stereotype.Repository;

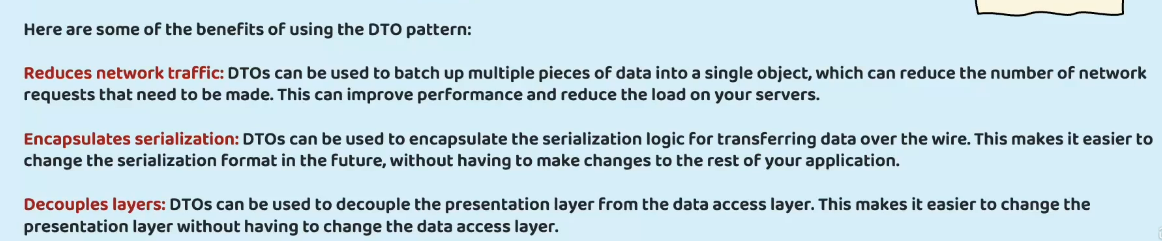
import com.eazybytes.accounts.entity.Accounts;

@Repository

public interface AccountsRepository extends JpaRepository<Accounts, String> {

}

**Introduction to DTO (data transfer objects)**So entity classes are used to create, update, delete accounts and customer detail or records inside a database.   
But what if a client application wants you accounts and customer details together in a single request. So, we cannot send 2 different objects inside a response, we can send only 1 single object inside a response.  
These entity classes are related to the database layer,   
 ****

****

**Creating DTO’s inside accounts microservice**create package com.eazybytes.accounts.dto  
  
**@Data** this not only creates the getters and setters but also the hashcode and equals methods. But this does not create a constructor which accepts all fields as arguments.   
  
package com.eazybytes.accounts.dto;

import lombok.Data;

@Data

public class AccountsDto {

private Long accountNumber;

private String accountType;

private String branchAddress;

}

package com.eazybytes.accounts.dto;

import lombok.Data;

@Data

public class CustomerDto {

private String name;

private String email;

private String mobileNumber;

}

package com.eazybytes.accounts.dto;

import lombok.AllArgsConstructor;

import lombok.Data;

@Data @AllArgsConstructor

public class ResponseDto {

private String statusCode;

private String statusMsg;

}

package com.eazybytes.accounts.dto;

import java.time.LocalDateTime;

import org.springframework.http.HttpStatus;

import lombok.AllArgsConstructor;

import lombok.Data;

@Data @AllArgsConstructor

public class ErrorResponseDto {

private String apiPath;

private HttpStatus errorCode;

private String errorMsg;

private LocalDateTime errorTime;

}